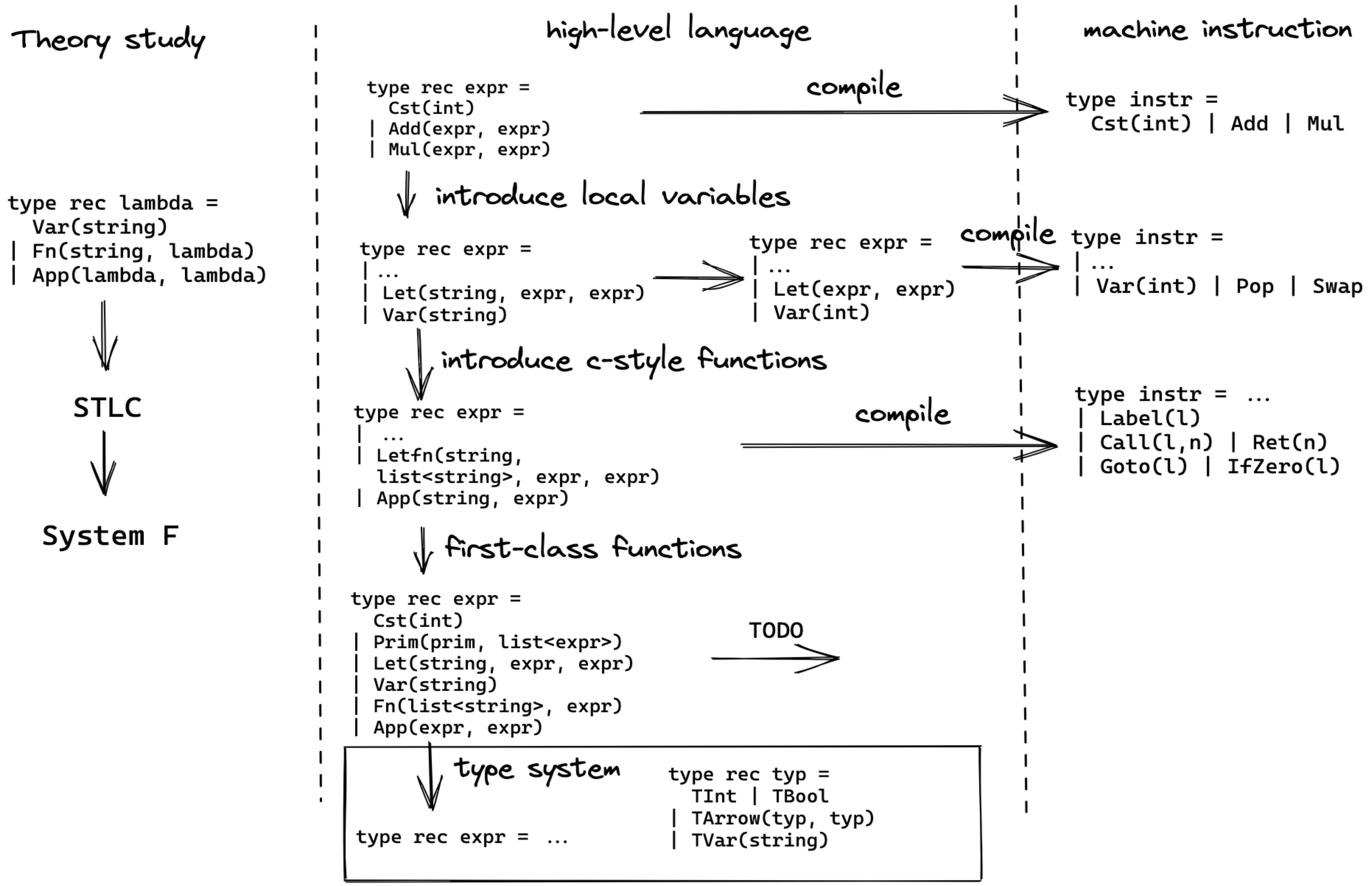
语义分析与类型(Part2)

基础软件理论与实践公开课

ZhangYu

idea 基础软件中心



Theory study
high-level language
machine instruction
type rec expr =
Cst(int)
| Add(expr, expr)
| Mul(expr, expr)
compile
type instr =
Cst(int) | Add | Mul
introduce local variables
type rec lambda =
Var(string)
| Fn(string, lambda)
| App(lambda, lambda)
type rec expr =
| ...
| Let(string, expr, expr)
| Var(string)
type rec expr =
| ...
| Let(expr, expr)
| Var(int)
compile
type instr =
| ...
| Var(int) | Pop | Swap
STLC
introduce c-style functions
type rec expr =
| ...
| Letfn(string,
list<string>, expr, expr)
| App(string, expr)
compile
type instr = ...
| Label(l)
| Call(l,n) | Ret(n)
| Goto(l) | IfZero(l)
System F
first-class functions
type rec expr =
Cst(int)
| Prim(prim, list<expr>)
| Let(string, expr, expr)
| Var(string)
| Fn(list<string>, expr)
| App(expr, expr)
TODO
type system
type rec expr = ...
type rec typ =
TInt | TBool
| TArrow(typ, typ)
| TVar(string)

Introduction

- Review: semantics analysis
  - Semantics analysis prevents run-time errors
  - Examples: using a variable not in scope, adding int with function, etc.
  - Type safety and type checking
- Overview: type inference
  - Constraint-based type inference
  - A more efficient implementation

Tiny language with types

- Types

```
type rec typ = TInt | TBool | TVar(string) | TArr(typ, typ)
```

- Expressions

```
type rec expr = CstI(int) | CstB(bool) | Var(string)
                | If(expr, expr, expr)
                | Fun(string, expr) | App(expr, expr)
```

`Prim` and `Let` are missing: think how to support them

- Overall goal

```
let infer = (expr: expr) : typ => { ... }
```

Intuition

For example, we want to infer the type of

$$\lambda f.\, \lambda a.\, \lambda b.\, \textsf{if } a \textsf{ then } f(b) + 1 \textsf{ else } f(a)$$

- Insert **type variables**

$$\lambda f : X.\, \lambda a : Y.\, \lambda b : Z.\, \textsf{if } a \textsf{ then } f(b) + 1 \textsf{ else } f(a)$$

- Generate constraints
  - from $f(a)$, we can infer $X = Y \to T_1$ for some T_1
  - from $f(b) + 1$, we can infer $X = Z \to \textsf{Int}$
  - from $\textsf{if} \ldots \textsf{then} \ldots \textsf{else} \ldots$, we can infer $Y = \textsf{Bool}$ and $T_1 = \textsf{Int}$
- After solving the constraints, we have $(\textsf{Bool} \to \textsf{Int}) \to \textsf{Bool} \to \textsf{Bool} \to \textsf{Int}$

Implementation

- Type variables and constraints

```
type constraints = list<(typ, typ)>
```

- Collect constraints: $\Gamma \vdash t : (T, C)$

```
let check_expr = (ctx: context, expr: expr) : (typ, constraints) => { ... }
```

- Solve constraints

```
type subst = list<(string, typ)>
let solve = (cs: constraints) : subst => { ... }
```

- Apply substitution

```
let type_subst = (t: typ, s: subst): typ => { ... }
```

Constraint Generation Rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : (T, \varnothing)}\text{C-Var} \qquad \frac{}{\Gamma \vdash i : (\mathsf{Int}, \varnothing)}\text{C-Int} \qquad \frac{}{\Gamma \vdash b : (\mathsf{Bool}, \varnothing)}\text{C-Bool}$$

$$\frac{\text{fresh } T \quad \Gamma \vdash t_1 : (T_1, C_1) \quad \Gamma \vdash t_2 : (T_2, C_2) \quad \Gamma \vdash t_3 : (T_3, C_3)}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : (T, C_1 \cup C_2 \cup C_3 \cup \{T_1 = \mathsf{Bool}, T = T_2, T = T_3\})}\text{C-If}$$

基础软件中心

Constraint Generation Rules

$$\frac{\text{fresh } T \qquad \Gamma, x : T \vdash t : (T_1, C)}{\Gamma \vdash \lambda x.\, t : (T \to T_1, C)} \text{C-Abs}$$

$$\frac{\text{fresh } T \qquad \Gamma \vdash t_1 : (T_1, C_1) \qquad \Gamma \vdash t_2 : (T_2, C_2)}{\Gamma \vdash t_1\ t_2 : (T, C_1 \cup C_2 \cup \{T_1 = T_2 \to T\})} \text{C-App}$$

Implementation

idea
基础软件中心

- syntax directed

```
let rec check_expr = (ctx: context, expr: expr): (typ, constraints) => {
  switch expr {
  | ...
  | Fun(x, e) => {
      let tx = new_tvar()  // TVar(fresh_name)
      let (te, c) = check_expr(list{(x, tx), ...ctx}, e)
      (TArr(tx, te), c)
    }
  | App(e1, e2) => {
      let t = new_tvar()  // TVar(fresh_name)
      let (t1, c1) = check_expr(ctx, e1)
      let (t2, c2) = check_expr(ctx, e2)
      let c = list{(t1, TArr(t2, t))}
      (t, List.concat(list{c1, c2, c}))
    }
  }
}
```

idea
基础软件中心

Example

Check the expression

$$\lambda f.\, \lambda a.\, \lambda b.\, \text{if } a \text{ then } f(b) + 1 \text{ else } f(a)$$

generates the type variables X, Y, Z, T_1, T_2, T_3 and the following constraints

```
(X, Z -> T_1)  // generated by f(b)
(T_1, Int)     // generated by f(b)+1
(X, Y -> T_2)  // generated by f(a)
(Y, Bool)        \
(Int, T_3)        |-> generated by if ... then ... else
(T_2, T_3)       /
```

and the result type `X -> Y -> Z -> T_3`

idea
基础软件中心

Solve constraints

```
let solve = (cs: constraints): subst => {
  let rec go = (cs, s): subst => {
    switch cs {
    | list{} => s
    | list{c, ...rest} =>
      switch c {
      | (TInt, TInt) | (TBool, TBool) => go(rest, s)
      | (TArr(t1, t2), TArr(t3, t4)) => go(list{(t1, t3), (t2, t4), ...rest}, s)
      | (TVar(x), t) | (t, TVar(x)) =>
        assert !(occurs(x, t)) // error report
        go( rest[t/x] , list{ (x, t), ...s }) // pseudocode!
      | _ => assert false // error report
      }
    }
  }
}
```

Example

Constraints that are unsolvable

- Try to unify `T_1 -> T_2` with `Bool`, etc
- Fail the occur check, such as `T_1` and `T_1 -> Int`

```
let occurs = (tvar: string, t: typ) : bool = { ... }
```

Example

Back to our example, we have the following constraints:

```
(X, Z -> T_1)  // generated by f(b)
(T_1, Int)     // generated by f(b)+1
(X, Y -> T_2)  // generated by f(a)
(Y, Bool)       \
(Int, T_3)       |-> generated by if ... then ... else
(T_2, T_3)      /
```

gives the substitution `X |-> Z -> T_1`

and the remaining constraints

```
(T_1, Int)
(Z -> T_1, Y -> T_2)
(Y, Bool)
(Int, T_3)
(T_2, T_3)
```

Example

Current substitution:

`X |-> Z -> T_1`

with the remaining constraints:

```
(T_1, Int)
(Z -> T_1, Y -> T_2)
(Y, Bool)
(Int, T_3)
(T_2, T_3)
```

New substitution (order matters!):

`X |-> Z -> T_1` `T_1 |-> Int`

with the remaining constraints

```
(Z -> Int, Y -> T_2)
(Y, Bool)
(Int, T_3)
(T_2, T_3)
```

idea
基础软件中心

Example

Current substitution:

`X |-> Z -> T_1` `T_1 |-> Int`

with the remaining constraints

```
(Z -> Int, Y -> T_2)
(Y, Bool)
(Int, T_3)
(T_2, T_3)
```

Substitution is unchanged:

`X |-> Z -> T_1` `T_1 |-> Int`

with the constraints reduced to

```
(Z, Y)
(Int, T_2)
(Y, Bool)
(Int, T_3)
(T_2, T_3)
```

Example

Current substitution:

`X |-> Z -> T_1` `T_1 |-> Int`

with the constraints:

```
(Z, Y)
(Int, T_2)
(Y, Bool)
(Int, T_3)
(T_2, T_3)
```

After two steps we have substitution:

`X |-> Z -> T_1` `T_1 |-> Int` `Z |-> Y` `T_2 |-> Int`

with the remaining constraints

```
(Y, Bool)
(Int, T_3)
(Int, T_3)
```

and eventually we arrived at the substitution:

`X |-> Z -> T_1` `T_1 |-> Int` `Z |-> Y` `T_2 |-> Int` `Y |-> Bool` `T_3 |-> Int`

Example

Applying the substitution

`X |-> Z -> T_1` `T_1 |-> Int` `Z |-> Y` `T_2 |-> Int` `Y |-> Bool` `T_3 |-> Int`

to

`X -> Y -> Z -> T_3`

gives us the result

`(Bool -> Int) -> Bool -> Bool -> Int`

idea
基础软件中心

Implementation

```
let infer = (expr: expr) : typ => {
  let (t, cs) = check_expr(list{}, expr)
  let s = solve(cs)
  let res = type_subst(t, s)
  res
}
```

- Homework: complete the type inference (two substitution functions)

What's the problem of our implementation?

- Quadratic time complexity in the number of constraints
- Error report is unfriendly

Can we improve it?

- Constraints and substitutions (which are slow) are used to represent equivalence
- Main idea: maintain the equivalence classes directly
- Union-find data structure can help

Example

- For example,

$$\lambda f.\, \lambda a.\, \lambda b.\, \textsf{if } a \textsf{ then } f(b) + 1 \textsf{ else } f(a)$$

- The constraints from the previous example:

```
(X, Z -> T_1)     (Y, Bool)
(T_1, Int)        (Int, T_3)
(X, Y -> T_2)     (T_2, T_3)
```

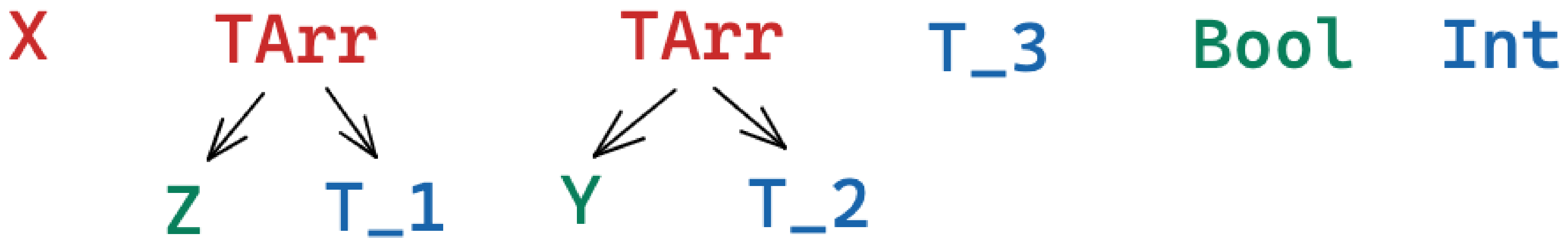
- What are the equivalence classes?

Example

- The constraints from the previous example:

```
(X, Z -> T_1)     (Y, Bool)
(T_1, Int)        (Int, T_3)
(X, Y -> T_2)     (T_2, T_3)
```

- Each color represents an equivalence class:



- If we have build such structure, then what is the `X -> Y -> Z -> T_3`?

Type inference

- Main idea: build such structure while traversing the AST
- Always pick function types or `Int` or `Bool` as representatives
- What happens if a function type and an `Int` or `Bool` are in the same equivalence class?
- Implementation: replace the constraints with unification

Implementation

Recall the typing rule

$$\frac{\text{fresh } T \qquad \Gamma \vdash t_1 : (T_1, C_1) \qquad \Gamma \vdash t_2 : (T_2, C_2)}{\Gamma \vdash t_1\ t_2 : (T, C_1 \cup C_2 \cup \{T_1 = T_2 \to T\})}\text{C-App}$$

```
let rec check_expr = (ctx, expr: expr) => {
  switch expr {
  | ...
  | App(e1, e2) => {
      // TVar(Nolink(fresh_name))
      let t = new_tvar()
      let t1 = check_expr(ctx, e1)
      let t2 = check_expr(ctx, e2)
      // unify the two types
      unify(t1, TArr(t2, t))
      t
    }
  }
}
```

```
let rec check_expr = (ctx, expr: expr) => {
  switch expr {
  | ...
  | App(e1, e2) => {
      // TVar(fresh_name)
      let t = new_tvar()
      let (t1, c1) = check_expr(ctx, e1)
      let (t2, c2) = check_expr(ctx, e2)
      // new constraint
      let c = list{(t1, TArr(t2, t))}
      (t, List.concat(list{c1, c2, c}))
    }
  }
}
```

Implement unification efficiently

- Union-Find data structure is used to track equivalence between objects
- Operations of union-find data structure
  - `new` : create a new node
  - `find (n)` : given a node `n`, find its representative
  - `union (n1, n2)` : make `n1` and `n2` equivalent
- Corresponding operations in type system:

```
let new_tvar = () : typ => { ... }
let type_repr = (t: typ): typ => { ... }
let unify = (t1: typ, t2: typ): unit => { ... }
```

idea
基础软件中心

Implementation

- To track the equivalence, type variables can be linked to some other types

```
type rec typ = TInt | TBool | TArr(typ, typ) | TVar(ref<tvar>)
and tvar = Nolink(string) | Linkto(typ)
```

- Create a new type variable

```
let new_tvar = () => TVar(fresh_name())
```

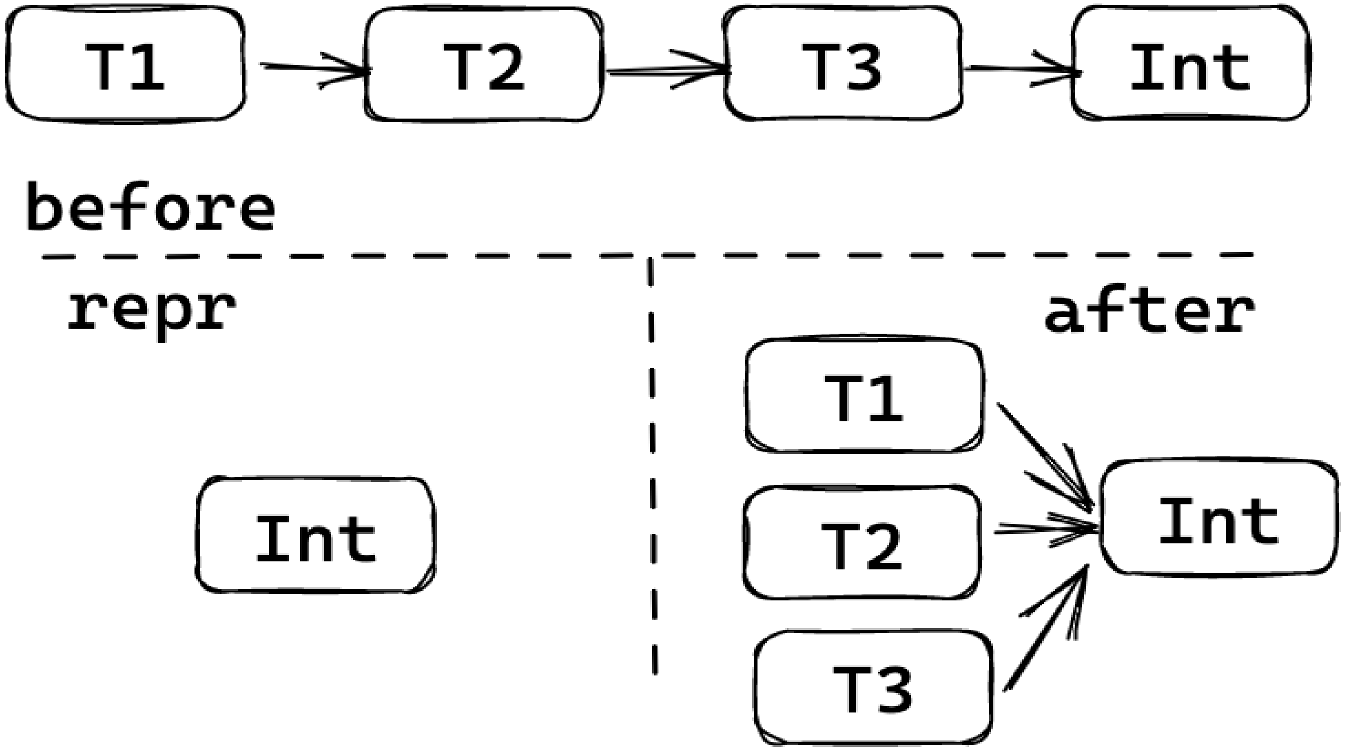
Implementation

- Find the representative: use the path compression trick

```
let rec repr_type = (t: typ): typ => {
  switch t {
  | TVar(tvar: ref<tvar>) =>
    switch tvar.contents {
    | Nolink(_) => t
    | Linkto(t1) => {
        let t1' = repr_type(t1)
        tvar := Linkto(t1')  // Side effect: path compression!
        t1'
      }
    }
  | _ => t
  }
}
```

Example of path compression

Call `repr_type` on `T_1`:



Note the arrows here are **links** rather than arrow types!

Implementation

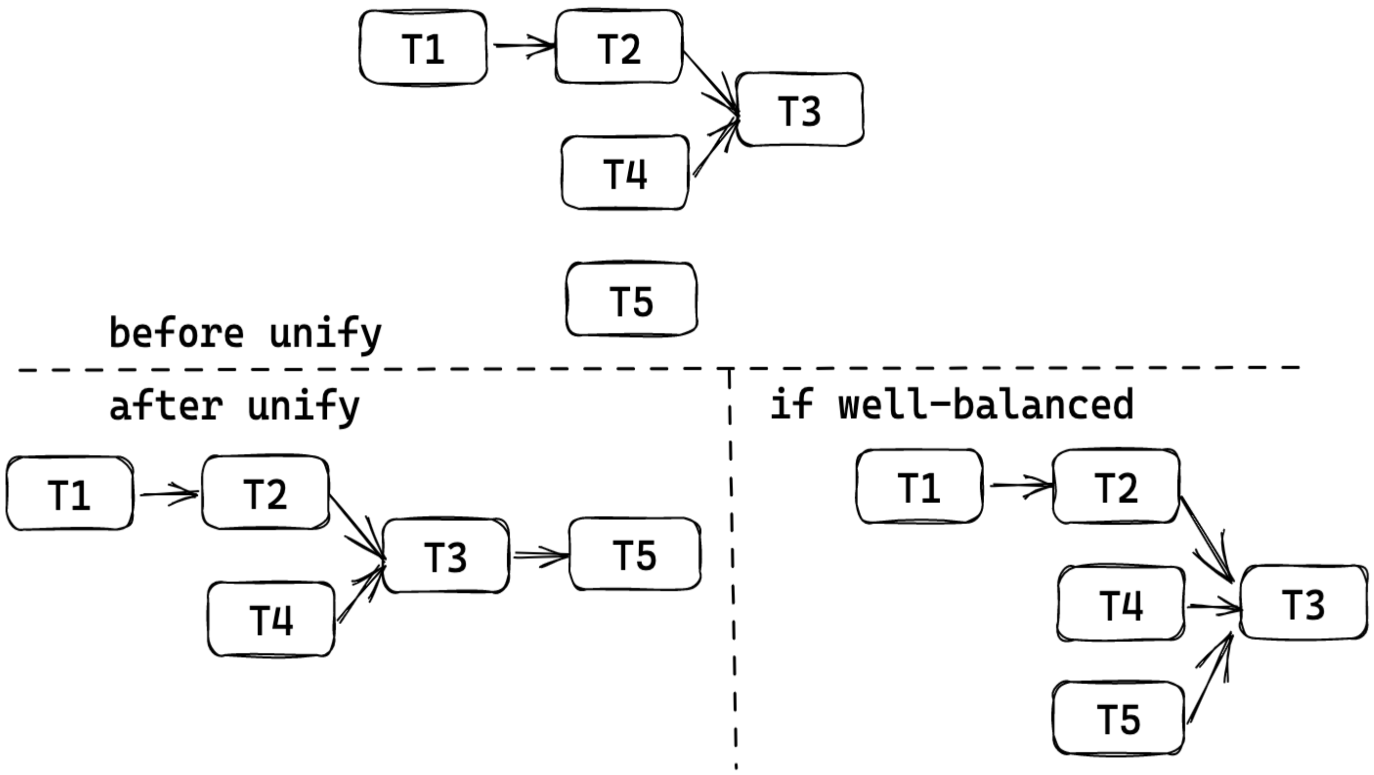
idea
基础软件中心

- Unification of two types

```
let rec unify = (t1: typ, t2: typ): unit => {
  let t1' = type_repr(t1) and t2' = type_repr(t2)
  if t1' === t2' { () }
  else
    switch (t1', t2') {
    | (TInt, TInt) | (TBool, TBool) => ()
    | (TArr(t1, t2), TArr(t3, t4)) => {
        unify(t1, t3)
        unify(t2, t4)
      }
    | (TVar(tvar), t) | (t, TVar(tvar)) =>
      assert !(occurs(tvar, t)) // error report
      tvar := Linkto(t)
    | _ => assert false // error report
  }
}
```

Example of unification

Call `unify` on `T4` and `T5`



Summary

- Constraints-based type inference
- Nore efficient implementation using union-find

Next class:

- Let-polymorphism
- Type schemes, generalization, and instantiation

Recommended reading and references

[1] Section 22 in Types and Programming Languages

[2] Section 6 in Programming Language Concepts

[3] Caml Light source code